Building AI-Intensive Software *with* AI: Early Results and a Cautionary Tale on Measuring Development Cost

Victor Barros de Miranda Neves
Centro de Informática, Universidade Federal de Pernambuco
Recife, Brasil
vbmn@cin.ufpe.br

Kiev Santos da Gama
Centro de Informática, Universidade Federal de Pernambuco
Recife, Brasil
kiev@cin.ufpe.br

Vinicius Cardoso Garcia
Centro de Informática, Universidade Federal de Pernambuco
Recife, Brasil
vcg@cin.ufpe.br

Abstract

Empirical reports on the true cost of AI-intensive software development remain scarce, and the few that exist are easy to get wrong in ways that never surface in the final number. We report early results from an ongoing case study: a six-person student team built a full conversational onboarding assistant — RAG-based code chat, guided tours, dependency graphs, technical-debt analysis — over one academic term using pervasive AI assistance. We instrumented development with a three-layer cost model (real AI spend, self-reported human effort, human counterfactual) and initially reported a 19.4× cost ratio. A follow-up pass revealed two independent errors — inferring per-token cost under a flat-rate subscription, and pricing the counterfactual with the wrong regional labor rates — that together had inflated the ratio by roughly 2×; the corrected figure is ∼9.9×. We present this correction as an early, generalizable finding in its own right: both errors are easy to make, invisible in the final number, and plausibly common in similar reports. We outline next steps toward a more robust, replicable costing methodology for AI-intensive development.

Keywords

AI-Assisted Software Development, Generative AI, Software Economics, Retrieval-Augmented Generation, Developer Onboarding

1 Introduction

Large language models now execute complex software engineering tasks end-to-end — writing code, generating tests, drafting documentation — but most evidence evaluates isolated tasks rather than complete systems built under pervasive AI assistance [4, 8]. Coding assistants have moved from single-line completion to agentic modes that plan, edit multiple files, and iterate, but open questions remain around correctness, review burden, and the economics of adoption [12]. Broader reviews of AI for software engineering explicitly call for more evidence on real-world adoption and outcomes [1], and this call is sharpest for cost: practitioners lack empirical baselines for how much effort can realistically be delegated to an AI agent, and the few cost figures that circulate [5, 11] are rarely audited for measurement error.

The setting is further complicated when the system under construction is itself AI-intensive. Building a tool that heavily relies on LLMs and retrieval pipelines creates a doubly recursive scenario: engineers use AI to build software whose core behavior also depends on AI. We chose to study this scenario through a concrete case: a conversational onboarding assistant that helps developers navigate legacy codebases via retrieval-augmented generation (RAG) [10], addressing a well-documented technical bottleneck in developer onboarding [2, 6, 9] and developer experience more broadly [3, 7].

We report emerging results from this ongoing case study. A six-person student team built the assistant over one academic term, relying almost entirely on AI assistance for both design and implementation. We instrumented the project with a three-layer cost model — real AI spend, self-reported human effort, and an estimated human counterfactual — as a first step toward a general costing methodology for AI-intensive development.

The most important finding was not the cost ratio itself but how fragile it turned out. The first analysis reported a 19.4× ratio, later corrected to 9.9× after we identified two independent, easy-to-miss errors. We present this correction, our methodology, and early lessons as preliminary evidence for a broader claim we are continuing to investigate: that cost claims about AI-assisted development deserve more scrutiny than the literature currently gives them.

2 Background and Related Work

AI-assisted software development. The use of LLMs to assist software engineering tasks has grown rapidly, and recent surveys map an expanding landscape of applications from requirements to maintenance [4, 8]. Coding assistants operating at the task level, instead of merely providing single-line completions, introduced "agent" modes that plan, edit multiple files, run commands, and iterate. While reported benefits center on implementation throughput, open questions remain around correctness, review burden, and the economics of adoption [12]. Because most of this literature evaluates model capability on benchmark tasks, our work provides a complementary perspective by analyzing the process and cost of building a complete system pervasively using AI assistance.

Program comprehension and developer onboarding. Understanding an unfamiliar codebase is a long-standing bottleneck. Industrial tools target parts of this problem, but they typically focus on pointwise questions or require manual authoring of walkthrough content. Research combining program comprehension with LLMs demonstrates that models can explain code and answer repository questions when fed adequate context via RAG [10]. Codebase health is also a recognized and actionable factor in developer experience [7]. Onboarding literature often focuses on social integration in agile settings [2, 6], but our domain specifically addresses the technical axis of onboarding. The system described in this paper serves as the engineering case study to ground our emerging results.

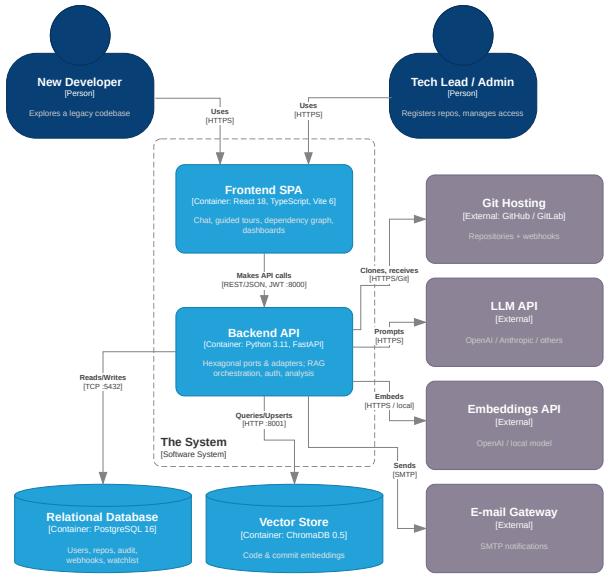

Figure 1: C4 container diagram: a React client calls a FastAPI backend (hexagonal ports and adapters) that orchestrates a vector store, relational DB, and external Git, LLM, embedding, and e-mail services. Edge labels give protocols and ports.

Table 1: Scale of the artifact. Line counts via `cloc` over hand-written source only (excluding dependencies, generated code, and configuration).

Dimension	Value
Features delivered	25+ across 4 phases
Supported languages	15 (via tree-sitter)
Automated tests	201 cases in 26 files (unit, integration, e2e)
Backend	Python, FastAPI, hexagonal architecture
Frontend	React, Vite, TypeScript
Retrieval	Vector store + sentence/OpenAI embeddings (RAG)
LLM providers	Configurable (Anthropic / OpenAI / others)
Lines of code	~21,000 production; ~2,000 tests

Cost Measurement in Practice. Broad reviews of AI for software engineering call for more evidence on real-world adoption and outcomes [1]. Transparent, audited cost measurements of AI-assisted development remain uncommon, particularly for AI-intensive products built in resource-constrained academic settings. Existing evidence rarely audits its own measurement pipeline: productivity studies report time or throughput gains rather than dollar cost [11], and cost-focused studies report top-line percentages without exposing pricing or counterfactual assumptions [5]. Our case study fills this gap by treating measurement error itself as a finding.

3 Case and Method

The system. The team built a conversational onboarding assistant addressing a recognized gap where onboarding literature shows codebase orientation delivered largely through mentoring and ad

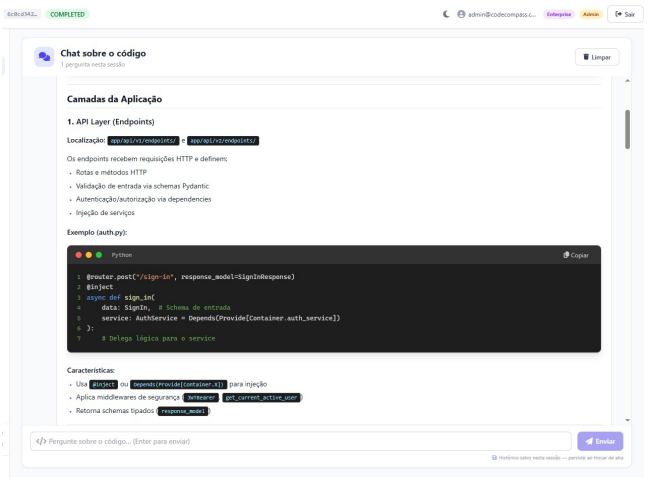

Figure 2: The RAG chat in use (UI in PT-BR). Answers are grounded in real repository fragments (here a router declaration, dependency injection, and a typed schema) rather than generic model knowledge.

hoc peer support [3, 7, 9], drawing experienced developers away from their own work. The tool clones a Git repository, chunks source code with a multi-language parser (tree-sitter, 15 languages), embeds code and commit history into a vector store, and powers a RAG-based chat, automated guided tours, a module dependency graph, and technical-debt analysis grounded in real repository fragments rather than generic model knowledge. The backend follows a hexagonal architecture (Python/FastAPI); the frontend is a React single-page application. Over one academic term, the team delivered 25+ features and 201 automated tests (~21,000 production lines of code, ~2,000 test lines).

Development process. Work proceeded through four phases (problem framing, solution design, build-and-test, and measurement) using two AI tools: a frontier-model LLM assistant for early-phase design research, and a mid-tier AI coding agent for implementation, testing, and documentation.

Three-layer cost model. For each phase, we recorded: (1) real AI spend, taken from provider billing records (a flat-rate coding-agent subscription plus metered charges for repository analysis and embeddings); (2) self-reported human effort, hours logged by team members; and (3) a human counterfactual, a bottom-up estimate of the hours a professional of a given seniority (junior/mid/senior) would need to complete each recorded task without AI assistance, priced at regional hourly rates (US$3.86/6.67/10.88) derived from local gross salaries. Seniority was assigned by task complexity, not by which student performed the work. The effort and counterfactual figures are self-reported estimates subject to hindsight bias.

4 Results and Findings

Comparing the project's actual cost with its estimated cost without AI reveals a large gap, but the key question is where that gap comes from, and where our initial estimation went wrong.

Cost across phases. Table 2 consolidates the three layers. Tooling cost US$69 in total: a flat GitHub Copilot Business subscription

(US$19/month over three months, apportioned across phases), a metered repository-analysis service (US$12.13), and embedding generation (US$0.13). The team logged 35.2 hours of human effort. Valuing that effort at the mid level rate gives US$235, for a total project cost of *US$304*. The counterfactual (329 professional hours at the seniority profile each activity demands) comes to *US$3,005*. The resulting ratio is *∼9.9*×.The distinction between the two effort figures matters and is easy to conflate: the 329 hours are an estimate of what a professional team would have needed *without* AI; the team's own logged effort was 35.2 hours, roughly nine times less.

Adjustments to the Cost Model. Our first analysis of these data reported a ratio of 19.4× and identified the problem-framing phase as consuming 68% of the AI budget, which we interpreted as evidence that model selection dominates the economics of AI-assisted development. Both claims were wrong, and the reasons are instructive enough that we report them rather than quietly correcting them. The error had two parts. *First*, we estimated AI cost by counting tokens and multiplying by published per-token rates for the model we believed the coding agent was using. It was not: the agent ran under a *flat-rate* subscription, so token volume had no effect on what we paid. The US$22.74 we attributed to the problem-framing phase. It appeared to consume two-thirds of the budget and led us to conclude that model choice drove cost—was an artifact of that assumption: a plausible figure computed from a pricing model that did not apply. Once billing records replaced estimates, tooling cost ∼US$69 in total, distributed in proportion to each phase's duration instead of which model it used. *Second*, we priced the counterfactual using hourly rates from national salary surveys skewed toward the largest metropolitan markets, roughly 85% above rates in the region where the work was performed. Correcting to regional rates lowered the counterfactual from US$5,678 to US$3,005. The two corrections move in the same direction—both had inflated the apparent advantage of AI—and together they take the ratio from 19.4× to 9.9×. We think 9.9× is the more useful number precisely because every term in it is now traceable to a billing record or a stated rate.

Distribution of AI-assisted work. The team's self-assessed reliance on AI varied sharply by activity, and the gradient is itself the finding. Implementation-level work was mostly delegated: code writing and test generation were about 95% AI-assisted, and debugging around 80%. Work that pairs mechanical output with human intent sat in the middle: documentation near 85% and requirements analysis near 70%. Judgment-heavy work stayed predominantly human—prompt design was only about 40% AI-assisted and architectural decisions about 20%. Thus, AI removed implementation friction while the decisions shaping the system remained with the team; the tool amplified the engineers instead of replacing them.

Benefits and Limitations of AI Assistance. Qualitatively, the AI assistance was key to rapidly exploring unfamiliar domains and translating established design decisions into consistent, tested code, whereas the assistance introduced friction in three conditions: (1) the models occasionally generated overly optimistic effort estimates; (2) long agent sessions silently lost context, forcing the re-exploration of previously read files; and (3) implementing features directly from prompts without a written design led to significantly more rework compared to features built from explicit specifications.

5 Discussion

5.1 Lessons Learned

Pricing model verification precedes cost optimisation. Our costliest error was not a cost but a misunderstanding of how cost was incurred: we assumed per-token billing for a premium model, when the coding agent in fact ran under a flat monthly subscription, where token volume has no effect on spend. Optimising token usage under a flat-rate plan is wasted effort; a metered service, conversely, requires exactly that discipline. We suggest establishing for each tool whether it is flat-rate, metered, or hybrid before optimising, reading billing dashboards rather than inferring cost from token counts, and recording metered charges as they occur rather than reconstructing them retrospectively.

Explicit context management in long agent sessions. Silent context loss was the most common source of wasted effort in our case: earlier context was compacted away without warning, prompting the agent to re-explore files or contradict earlier decisions. Two habits appeared to mitigate this: a short running state note before each subtask, and recording architectural decisions in repository files rather than relying on conversational continuity.

Specification-first task definition. In our observation, features implemented directly from conversational requests required more iterations than those based on explicit specifications; without a written design, the agent filled gaps with plausible but often incorrect assumptions. This suggests AI assistance amplified existing clarity rather than substituting for it.

Prompt cataloguing as an engineering practice. Treating core application prompts as versioned artifacts with a fixed template appeared to aid reuse and review, giving them a level of scrutiny comparable to source code.

Security practices established from initial development. The pace of AI-assisted development makes security debt easy to accumulate. We adopted upfront rules (e.g., no hard-coded credentials in code or agent transcripts, constant-time comparison for webhooks, centralized audit logging) as a precaution against this risk.

Interpretation of the reported cost ratio. A ratio of 9.9× is not a controlled measurement of productivity: it divides a numerator now grounded in billing records and time logs by a denominator that remains a retrospective estimate of work never performed. It is also a conservative figure, since the 329 counterfactual hours assume a senior developer already fluent in the stack; a mid-level profile would plausibly raise the ratio to 13–16×. We report the lower estimate deliberately and treat the multiplier as an order-of-magnitude signal from a single case, not a benchmark.

5.2 Practical Implications

For SE Practice. *AI reallocated tasks, not headcount, in this project.* Delegation was highly uneven: implementation was almost fully AI-assisted, while architecture and prompt design was human-driven. Organizations may benefit from planning which activities to delegate rather than how many people to remove, with a *small team* potentially absorbing more implementation work than headcount alone would suggest, provided judgment-heavy roles remain staffed. AI accelerated verbose more than conceptually complex tasks, a

Table 2: Cost model per phase. Tool costs from billing records (flat-rate Copilot apportioned by usage); human effort self-reported at the mid-level rate (US$6.67/h); counterfactuals are retrospective estimates. Columns figures were rounded.

Phase	Tools (US$)	Human (h)	Human (US$)	Total w/ AI (US$)	Counterfact. (h)	Counterfact. (US$)	Ratio
Pre-proposal	6	7.0	47	53	31	256	4.9×
P1 (problem framing)	18	11.4	76	94	136	1,302	13.9×
P2 (solution design)	11	4.2	28	39	19	186	4.8×
P3 (build-and-test)	11	3.8	25	36	27	226	6.3×
P4 (measurement)	12	5.8	39	51	67	598	11.8×
Quality & improvements	12	3.0	20	32	49	436	13.6×
Total	**69**	**35.2**	**235**	**304**	**329**	**3,005**	**9.9×**

pattern story points may not capture. Finally, lightweight Architectural Description Records appeared to reduce context loss during long agent sessions, although we do not claim causation.

For SE Education. Because this project originated in an AI-assisted software engineering course, it suggests, but does not validate, curricular directions. Classical engineering practices, such as writing specifications, documenting decisions, and critically reviewing AI outputs, appeared to reduce rework. Having students record costs, effort, and counterfactuals also exposed measurement biases, suggesting pedagogical value beyond the software itself.

5.3 Limitations

As emerging results from a single, ongoing case study, these findings are preliminary. The counterfactual estimates effort that was never actually performed, based on retrospective student judgment rather than a measured baseline — the same class of error that produced our initial 19.4× ratio, so even 9.9× may not be final. The team consisted of students, not the professionals used to price the counterfactual, and effort is self-reported. As a single case, the ratio is an early signal to be tested across more projects, not a benchmark. We ground verifiable terms (billing records, time logs) where possible, but the number and the methodology are work in progress.

6 Conclusion

These early results suggest that pervasive AI assistance can shift the feasibility frontier for small development teams building non-trivial software: 35.2 hours of human effort and US$69 of tooling produced a system that we estimate would otherwise have taken 329 professional hours. We suspect the measurement errors we corrected, inferring flat-rate costs from tokens and misapplying distant labor rates, are common, making our corrected methodology as valuable as the 9.9x ratio itself.

Artifact Availability

A replication package[1] includes the development log, counterfactual references, redacted billing records, prompts, and documented limitations (self-reported effort, estimated counterfactual hours, and apportioned subscription costs).

Generative AI Use Disclosure

The authors disclose that generative AI tools assisted in drafting text, structuring tables, and reformulating language. The authors reviewed, edited, and verified all generated text and tables. Additionally, the software system described in this paper was developed using pervasive AI assistance, as documented throughout the study.

[1]Available at: https://doi.org/10.5281/zenodo.21843234

Acknowledgements

The authors thank the support of INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0.

References

[1] Iftekhar Ahmed et al. 2025. Artificial Intelligence for Software Engineering: The Journey So Far and the Road Ahead. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025). doi:10.1145/3719006

[2] Jim Buchan, Stephen G. MacDonell, and Jennifer Yang. 2019. Effective team onboarding in Agile software development: techniques and goals. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–11. doi:10.1109/ESEM.2019.8870189

[3] Fabian Fagerholm and Jurgen Munch. 2012. Developer experience: Concept and definition. In *2012 International Conference on Software and System Process (ICSSP)*. IEEE, 73–77. doi:10.1109/ICSSP.2012.6225984

[4] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. doi:10.1109/ICSE-FoSE59343.2023.00008

[5] Nuruzzaman Faruqui, Priyabrata Thatoi, Rohit Choudhary, Ivana Roncevic, Hamed Alqahtani, Iqbal H Sarker, and Shapla Khanam. 2024. AI-analyst: An AI-assisted SDLC analysis framework for business cost optimization. *IEEE Access* 12 (2024), 195188–195203.

[6] Peggy Gregory, Diane E. Strode, Helen Sharp, and Leonor Barroca. 2022. An onboarding model for integrating newcomers into agile project teams. *Information and Software Technology* 143 (mar 2022), 106792. doi:10.1016/j.infsof.2021.106792

[7] Michaela Greiler, Margaret-Anne Storey, and Abi Noda. 2023. An Actionable Framework for Understanding and Improving Developer Experience. *IEEE Transactions on Software Engineering* 49, 4 (apr 2023), 1411–1425. doi:10.1109/TSE.2022.3175660

[8] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024). doi:10.1145/3695988

[9] An Ju, Hitesh Sajnani, Scot Kelly, and Kim Herzig. 2021. A Case Study of Onboarding in Software Teams: Tasks and Strategies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 613–623. doi:10.1109/ICSE43902.2021.00063

[10] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help with Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. doi:10.1145/3597503.3639187

[11] Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Vahid Meimand, Nan Zhang, Ben Ferrari-Church, and Satish Chandra. 2025. How much does AI impact development speed? An enterprise-based randomized controlled trial. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 618–629.

[12] Jaakko Sauvola, Sasu Tarkoma, Mika Klemettinen, Jukka Riekki, and David Doermann. 2024. Future of Software Development with Generative AI. *Automated Software Engineering* 31 (2024). doi:10.1007/s10515-024-00426-z